An Approach to Stego-Container Organization in FPGA Systems for Approximate Data Processing

Kostiantyn Zashcholkin[a], Oleksandr Drozd[a], Olena Ivanova[a], Ruslan Shaporin[a] and Mykola Kuznietsov[a]

[a] *Odessa National Polytechnic University 1, Ave. Shevchenko, Odessa, 65044, Ukraine*

Abstract

The paper is devoted to the development of steganographic approaches to checking the integrity of an FPGA (Field Programmable Gate Array) system based on preserving the basic functionality of stego containers and hiding both the hash sum and other control information as well as the very fact of their existence. The program code of an FPGA project with a LUT-oriented (Look-Up Table) architecture is proposed to be used to organize stego-containers when performing approximate calculations in floating-point formats. The development of this approach is based on the progressive dominance of hardware support in the processing of approximate data and the orientation of modern CAD systems to provide library circuit solutions for performing complete arithmetic operations. Floating-point formats round the computed result to the size of the operand. The use of these formats in FPGA systems creates structural redundancy in the schemes for implementing complete arithmetic operations and thus eliminates the influence of a number of LUTs on the rounded result. The code stored in these LUT units can be used to organize stego containers. A TCL-application developed for this purpose generates a circuit description for an analyzed FPGA system with a LUT-oriented architecture. The following application uses the resulting description to evaluate the effect of LUT units on the bits of the calculated result. Experiments carried out for iterative array multipliers implemented in FPGA projects made it possible to estimate the volume of stego-containers that can be used when performing approximate calculations in floating-point formats.

Keywords

Steganographic approach, FPGA, LUT-oriented architecture, stego-container, floating-point format, approximate data processing, complete arithmetic operation

1. Introduction

A significant part of modern computer systems contains programmable components. The functioning of such components is determined by the program code. The program code can be changed at any time, which will lead to a change in the operation of the system. On the one hand, this is an advantage, since at any stage of operation of the system, it is possible to update its program code or eliminate errors found in it. On the other hand, it carries the threat of malicious interference in the operation of a computer system by exploiting the program code [1].

To minimize these threats, a large number of methods, tools and organizational measures have been developed [2]. Among them, operational monitoring of program code integrity plays the most important role [3].

IntelITSIS’2021: 2nd International Workshop on Intelligent Information Technologies and Systems of Information Security, March 24–26, 2021, Khmelnytskyi, Ukraine
EMAIL: const-z@te.net.ua (K. Zashcholkin); drozd@ukr.net (O. Drozd); en.ivanova.ua@gmail.com (O. Ivanova); rshaporin@gmail.com (R. Shaporin); koliaodessa@ukr.net (M. Kuznietsov)
ORCID: 0000-0003-0427-9005 (K. Zashcholkin); 0000-0003-2191-6758 (O. Drozd); 0000-0002-4743-6931 (O. Ivanova); 0000-0003-4407-2367 (R. Shaporin); 0000-0002-3043-5924 (M. Kuznietsov)

Integrity monitoring is effective for two reasons. First, integrity monitoring can be performed at any stage of the system life cycle. Second, integrity monitoring can be easily combined with other approaches to ensuring the integrity of the program code.

Over the last decade, FPGA (Field Programmable Gate Array) chips with LUT-oriented architecture have taken an increasing share of the programmable component market [4]. These chips are competitors of microprocessors and microcontrollers. At the same time with the possibility of reprogramming, FPGA demonstrate superiority over microprocessors for tasks that require high computing performance. Because of this specificity, FPGA are often used as components of safety-related computer systems for industrial, military and aerospace purposes [5].

FPGA chips are programmed by loading low-level program code (bitstream) into the configuration memory of these chips. Due to the criticality and importance of using FPGA microcircuits, the integrity of their program code is monitored every time it is updated, and can also be monitored regularly [6].

Traditionally, integrity monitoring of the program code is carried out by double calculation [7, 8] and comparison of hash sums [9]. When preparing the program code for monitoring, a reference hash sum is calculated. At the time of monitoring, the reference hash sum is compared with the newly calculated hash sum. In these processes, the method and location of storing the reference hash sum significantly affects the possibility of monitoring falsification. An effective approach to storing monitoring data for FPGA program code is the steganographic approach [10, 11].

Within the framework of this approach, monitoring data (hash-sum) are embedded in the program code in the form of a digital watermark [12, 13]. This implementation is performed by equivalent transformations of the program code [14]. Therefore, the size of the program code, the operation and characteristics of the device are not changed.

The digital watermark (which contains the reference hash sum) is hidden from the outside observer and is indistinguishable from the program code. At the time of monitoring, the hidden reference hash can be extracted by an authorized person who has a stego key. Also, elements of the steganographic approach can be used to obfuscate the FPGA program code [15, 16].

Moreover, hash-sum hiding and obfuscation can be performed using a single system of equivalent transformations. The purpose of obfuscation in this case is to make it difficult to stegoanalysis [17, 18] aimed at detecting a digital watermark in the program code. For this reason, the development of approaches to steganographic hiding of integrity monitoring data is an important and relevant task.

2. Literature Review and Goal of the Paper

The steganographic approach to data protection is based on the imperceptible (not attracting attention) embedding of protected data into an information object – a stego container. The main conditions for such embedding are:

1. Immutability of the main functionality for stego container as a result of implementation;
2. Indistinguishability of the embedded data from the stego container data.

The greatest development is the direction of steganography [19], which uses multimedia files as stego containers: bitmap images, digital video, digitized sound. The basis of steganography methods in this direction is that multimedia data are analogous and approximate nature. Images, video and sound are originally analog phenomena, converted to binary representation as a result of the use of appropriate analog sensors and then digitizing their readings.

Due to this, the elementary units of multimedia stego containers (pixels for bitmap images and video, samples for digitized sound) have areas of different significance. The presence of these areas is due to unequal bits of the digital representation of analog data. The least significant bits of pixels and samples contribute very little to the quantitative equivalent of these values. Because of this, the distortion of these bits does not affect the main function of the multimedia stego container. Such insignificant areas in the data of multimedia stego containers are manifested both in the spatial domain of their presentation and in the transformation domain. Methods of digital steganography use these insignificant areas as places for embedding hidden data.

Unlike multimedia stego containers, the FPGA code is not approximate data, but exact. It is impossible to change the bits of the FPGA program code by the methods used in multimedia steganography. This is due to the fact that such changes will lead to distortion of the functioning of the device. Because of this, steganographic methods based on equivalent transformations are used for FPGA program code [20]. These methods, in the process of data embedding, equivalently modify the values of the program code without affecting the operation of the device and its characteristics.

Thus, it can be stated that the following assignment of steganographic methods to the types of stego-containers has developed:

- for multimedia stego containers, methods for non-equivalent changes in the data of elementary container units;
- for containers with exactly represented data (for example, for FPGA program code) – methods for equivalent transformation of container data.

At the same time, the structure of the FPGA program code has the potential to perform non-equivalent embedding similar to those used for multimedia stego containers as well. Such embedding is possible for FPGA program code implementing arithmetic operations on approximate data.

Approximate data processing has long dominated precision calculations and continues to consolidate its dominance. This process can be traced on the example of the personal computer development, which shows an objective nature in the constant strengthening of hardware support for approximate calculations. The focus on approximate data processing begins with optional Intel 283/387 co-processors that support floating-point computations in hardware [21]. Then the floating-point data processing goes to the Intel 486DX central processor. The Pentium family offers the use of FPU (Floating-Point Unit) pipelines for accelerated processing of approximate data. A modern graphics processor contains thousands of FPU pipelines, which are used for parallel execution of approximate computations using CUDA technology [22].

Nobody planned such a development of personal computers. But this plan could be drawn up, since it is being implemented as a result of the natural process of structuring all types of resources for the peculiarities of the natural world. The history of the development of the computer world shows to the greatest extent the influence of two such features: parallelism and proximity, fuzziness of the natural world [23].

Following this development vector is motivated by a significant increase in important indicators. For several decades, personal computers have increased the clock frequency and, accordingly, the throughput from KHz to GHz. At the same time, the amount of memory increased from MB to TB. Thus, the main indicators of personal computers were simultaneously improved by a factor of millions.

Following the development vector that supports approximate computing can be observed in new directions of improving information technologies, including Big Data, Internet of Things and Everything, Cyber-physical and Safety-related systems, receiving initial data from sensors. These measurement results are approximate data [24-26].

Scientific directions are being developed, first of all, within the framework of the development of approximate models and methods that work under conditions of fuzziness [27-29], give and use approximate estimates of the significance of the information received and processed [30-32].

Thus, the development of steganography in the field of approximate data processing, including FPGA systems, is quite natural.

It should also be noted that this direction is supported by FPGA designing, which is still on the way to structuring under the fuzziness of the natural world. This feature is evident in CAD systems that support FPGA designing. They provide a wide range of library circuit solutions focused on performing complete arithmetic operations for accurate calculations and practically do not support the features of processing approximate data. The use of library circuits in approximate calculations demonstrates structural redundancy that can be used to organize stego-containers.

The presence of structural redundancy in circuit implementations of complete arithmetic operations is confirmed by the practice of using truncated calculations, which show an almost twofold simplification of the circuit without loss of accuracy in the case of performing the truncated operation with mantissas in an iterative array multiplier [33, 34].

Non-equivalent steganographic data embedding based on approximate computations and structural redundancy of FPGA systems can be combined with equivalent embedding, using version redundancy of LUT-oriented architecture [35, 36].

In this case, a hybrid scheme of steganographic data embedding into the FPGA program code appears. In this scheme, both embedding approaches (equivalent and non-equivalent) can play the same or different roles (hiding monitoring data, obfuscating of program code, false embedding). For example, one approach performs embedding to hide monitoring data, while the other is used to code obfuscation or for false embedding.

One of the conditions for the applicability of the proposed non-covalent implementation is to ensure that the volume of the stego container is sufficient for the indicated roles. Based on this, the *goal of this paper* is to estimate the volume of the stego container obtained by applying non-equivalent transformations to the FPGA program code.

3. Proposed approach to steganographic data embedding into FPGA program code

When performing arithmetic operations, it is typical to require results that are the same size as the operands. This requirement is common in floating point specifications. When performing an operation on *n*-bit operands, an *n*-bit result must be obtained. For a number of arithmetic operations, the size of the result naturally exceeds the size of the operands. In this case, the full result is calculated first. After that, rounding is performed and the lower bits are removed. As a result, the size of the result is made equal to the size of the operands.

For example, when performing the operation of multiplying *n*-bit mantissa of floating-point numbers, a complete 2*n*-bit result is obtained. This result is then rounded off by discarding the lower n bits. The remaining bits form the required *n*-bit result.

It should be noted that performing multiplication is a typical pattern for approximate calculations in general. Indeed, the representation and processing of approximate floating-point data is generally accepted and absolutely dominant. At the same time, the normal form of representation of numbers in these formats makes multiplication a key operation of approximate calculations, since it is used in the very record of floating-point data [37].

For example, one and a half million is represented as 1.5×10^6, where 1.5 is the mantissa, 10 is the base of the number system, 6 is the exponent. This representation determines the use of multiplication in all operations with mantissas in one form or another. In this case, the complete results of the operations inherit various properties of the product, including doubling the size of the operands in dual-operand operations.

For approximate calculations, the bits of the complete result can be divided into two subsets: retained (most significant) bits and discarded (least significant) bits [38]. Distortion in the least significant bits of the result does not affect the final result of the operation. When implementing such arithmetic operations on FPGA in its structure can be separated the elementary calculating units, which participate in the formation of only least significant (discarded) bits of the result. Due to this, the program code of such units may be non-equivalently modified and such modification will not affect the result of the device operation.

Further, the proposed approach and its evaluation are shown using the example of the multiplication operation. However, they can be extended to other arithmetic operations, in which the result is obtained by discarding the least significant bits.

Let $A = <a_{n-1}, a_{n-2}, \dots, a_0>$, $B = <b_{n-1}, b_{n-2}, \dots, b_0>$, $C = <c_{2n-1}, c_{2n-2}, \dots, c_n, c_{n-1}, c_{n-2}, \dots, c_0>$ be the operands and the complete result for the multiplication operation, respectively. To convert the result C to the required format (when specifying the same size of the operands and the result), the high-order n-bits of the result are retained and the low-order ones are discarded.

Thus, the bits of the complete result form two ordered sets:

$$C_{MSB} = < c_{2n-1}, c_{2n-2}, \dots, c_n>, C_{LSB} = < c_{n-1}, c_{n-2}, \dots, c_0> \quad (1)$$

where C_{MSB} – set of most significant bits;

C_{LSB} – set of least significant (discarded) bits.

The main elementary units of FPGA structure microcircuits are LUT (Look Up Table), which are generators of logical function of *m* variables, where *m* is the number of inputs of the LUT unit. Modern families of FPGA contain LUTs with the number of inputs from 4 to 8 and, accordingly, memory from 16 to 256 bits.

The LUT is programmed to calculate a specific logic function by a 2^m-bit program code. An FPGA chip that implements an arithmetic operation on approximate data, as a rule, is assembled from library modules designed to perform complete operations. Such chip contains LUT units that are used to calculate the least significant bits of the operation result and do not have any effect on the most significant bits:

$$L_{IE} \subset L, \tag{2}$$

where L – set of all LUTs in an FPGA project that implements arithmetic operations;

L_{IE} – a subset of LUTs that are involved in the calculating of least significant bits only and not involved in the calculating of most significant bits.

Hereinafter the LUTs included in the LIE set will be referred to as inessential LUTs. The program code of inessential LUTs can be non-equivalently modified in order to steganographically embed additional data into it. This modification will distort the result, but only in the least significant bits. Since these bits are discarded during rounding, the modification of the program code of L_{IE} units will not affect the final result, converted in the used format to the number of operand bits.

It is necessary to evaluate what volume of stego container provides the proposed non-equivalent approach to data embedding. For this purpose, an experimental evaluation is performed, in which the stego-containers are binary multipliers of different size, implemented on FPGA.

4. Experimental estimation of the stego container volume

The purpose of the experiment is to estimate the portion of inessential (included in the LIE set) LUT units in the total number of LUT units of the arithmetic device implemented on the FPGA. The multiplication operation is chosen as the target function of the device. This is due to the fact that this operation is key to performing approximate calculations.

4.1. The process of forming the initial data of the experiment

The initial data for the experiment were formed using the CAD system Intel Quartus Prime 20.1 Lite Edition [39, 40] and the software that we developed. The process of obtaining the initial data was as follows.

Five FPGA projects were created in the Intel Quartus Prime environment using library components. Each of these projects implements a multiplier of the corresponding size. For the experiment, we used iterative array multipliers with operands of 4, 6, 8, 12 and 16 bits. The multipliers have been configured to produce a complete $2n$-bit result from the n-bit operands. Thus, we were available the result that takes place before the bits are discarded.

Further, in the Intel Quartus Prime environment, synthesis was performed, as well as project placement and routing. The FPGA Intel Cyclone IV EP4CE15F23A7 [41] was selected as the target chip for the synthesis.

After placement and routing, a software application developed by us was used. This application is developed in TCL [42] and allows to interact with Intel Quartus Prime using the API (Application Programming Interface) [43] of this CAD system. TCL application extracts detailed information about the FPGA design structure from the internal Intel Quartus CAD database.

The extracted information includes:

a) a description of the LUTs included in the project;

b) program codes of each of the LUT units;

c) information about the connections of the LUTs with each other, as well as with the inputs and outputs of the chip.

The result of the functioning of the developed TCL application is the structure of the FPGA project presented in a form suitable for further analysis. This analysis was performed using the second application developed by us using the free programming environment Delphi 10 Seattle demo version [44].

4.2. The main part of the experiment

The main part of the experiment was implemented in the environment of the second application we developed. This application takes raw data (information about the structure of the FPGA project) and determines which of the project LUTs are inessential.

The process of analyzing LUT units is as follows. For each of the possible joint values of the multiplier operands *A* and *B*, the following steps are performed.

Step 1: The 2*n*-bit result of the multiplication of operands *A* and *B* is calculated.

Step 2: The correct values are fixed at the outputs of each of the FPGA project LUT units.

Step 3: The values at the output of each FPGA project LUT are inverted sequentially (changed from correct to incorrect). For each of these distortions, the value of the result error is recorded. The error value is calculated as a modulus of difference between the correct result value and the result value obtained after inversion.

Each of the LUT units of the project is associated with the maximum value of the error that arose as a result of the distortion of the output value of this unit. The stored error value is updated when the current error is greater than the stored error.

The values of the maximum error obtained as a result of the operand values enumeration allow to decide whether the LUT unit belongs to a subset of inessential units. For *n*-bit multiplier, if the maximum error corresponding to the LUT_i unit is less than 2^n, then this error appears only in the discarded bits of the result $C_{LSB} = <c_{n-1}, c_{n-2}, \dots, c_0>$.

This means that for any values of the operands, the distortion of the values at the output of the LUTi unit does not lead to the distortion of the most significant bits of the result.

Thus, the decision rule for assigning an n-bit multiplier LUT unit to a subset of inessential units is as follows:

$$\text{if } max\_error(LUT_i) < 2^n \text{ then } LUT_i \in L_{IE}, \qquad (3)$$

where $max\_error(LUT_i)$ – the maximum error value assigned to the unit LUT_i.

Fig. 1 shows the window of the second software application we developed. This window contains the experiment results for an 8-bit multiplier. In this window, for each LUT (1 ... 101) unit of the FPGA project, the value of the minimum and maximum errors obtained during the experiment is presented.

At the bottom of the window, you can see the statistics of errors that appear in each individual bit of the result.

Line B contains the bit numbers of the result from right to left, starting with the first least significant bit.

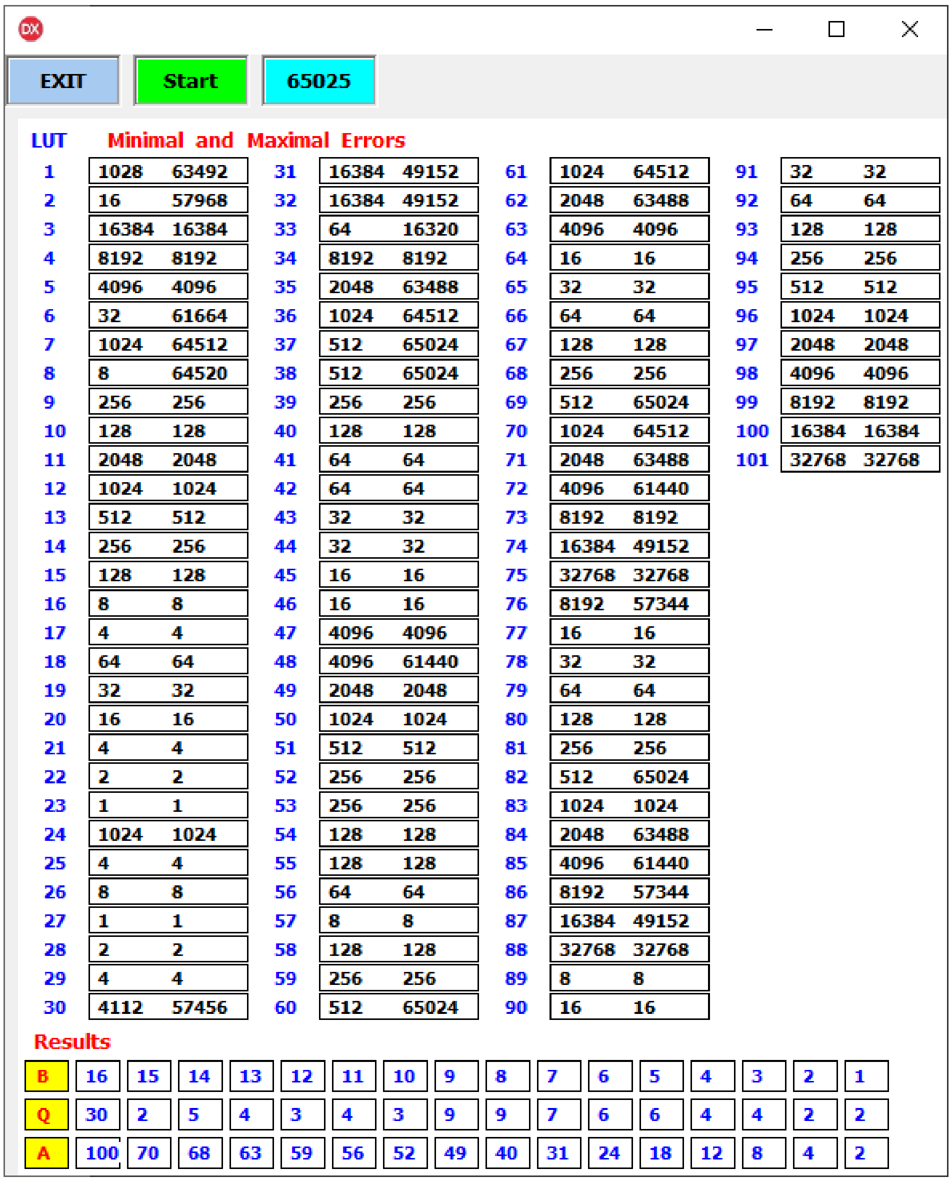
Line Q shows the number of LUT units affecting the calculated result in the bit with the specified number.

Line A shows the portion of LUT units whose influence on the result does not exceed the weight of the specified bit. For a given FPGA project, this estimate is nearly the same as the number of such LUT units.

4.3. Discussion of experimental results

The results of the experiment made it possible to obtain the number of inessential LUT units in multiplier circuits implemented on FPGA. Since the values at the outputs of inessential units do not affect the most significant bits (which remain after rounding) of the result, the program code of these units can be subjected to non-equivalent modification. Thus, the number of inessential units makes it possible to estimate the volume of the stego container formed in the environment of the LUT-circuit of the multiplier.

Table 1 shows the experimental results obtained for multipliers with operand sizes of 4, 6, 8, 12 and 16 bits.



LUT	Minimal and Maximal Errors						
1	1028 63492	31	16384 49152	61	1024 64512	91	32 32
2	16 57968	32	16384 49152	62	2048 63488	92	64 64
3	16384 16384	33	64 16320	63	4096 4096	93	128 128
4	8192 8192	34	8192 8192	64	16 16	94	256 256
5	4096 4096	35	2048 63488	65	32 32	95	512 512
6	32 61664	36	1024 64512	66	64 64	96	1024 1024
7	1024 64512	37	512 65024	67	128 128	97	2048 2048
8	8 64520	38	512 65024	68	256 256	98	4096 4096
9	256 256	39	256 256	69	512 65024	99	8192 8192
10	128 128	40	128 128	70	1024 64512	100	16384 16384
11	2048 2048	41	64 64	71	2048 63488	101	32768 32768
12	1024 1024	42	64 64	72	4096 61440		
13	512 512	43	32 32	73	8192 8192		
14	256 256	44	32 32	74	16384 49152		
15	128 128	45	16 16	75	32768 32768		
16	8 8	46	16 16	76	8192 57344		
17	4 4	47	4096 4096	77	16 16		
18	64 64	48	4096 61440	78	32 32		
19	32 32	49	2048 2048	79	64 64		
20	16 16	50	1024 1024	80	128 128		
21	4 4	51	512 512	81	256 256		
22	2 2	52	256 256	82	512 65024		
23	1 1	53	256 256	83	1024 1024		
24	1024 1024	54	128 128	84	2048 63488		
25	4 4	55	128 128	85	4096 61440		
26	8 8	56	64 64	86	8192 57344		
27	1 1	57	8 8	87	16384 49152		
28	2 2	58	128 128	88	32768 32768		
29	4 4	59	256 256	89	8 8		
30	4112 57456	60	512 65024	90	16 16		

Results

B	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Q	30	2	5	4	3	4	3	9	9	7	6	6	4	4	2	2
A	100	70	68	63	59	56	52	49	40	31	24	18	12	8	4	2

Figure 1: Experiment Results Window for 8-Bit Multiplier

Table 1
Experiment Results

Size of operands and final result	4	6	8	12	16
Size of the result before rounding	8	12	16	24	32
Total number of LUTs	30	61	101	208	341
Number of inessential LUTs	12	24	40	86	145
Portion of inessential LUT units among all units	40%	39,3%	39,6%	41,3%	42,5%
Minimum volume of stego container (bits)	12	24	40	86	145
Maximum volume of stego container (bits) when the number of inputs of LUT units is 4	192	384	640	1376	2320

In experimental FPGA projects, the portion of inessential LUT units ranged from 39.3% to 42.5%.

Table 1 also shows the upper and lower estimates of the stego container volume obtained by applying the proposed non-equivalent data embedding. The operation of an *m*-input LUT is specified by a 2^m bit program code (usually m is between 4 and 8). Thus, each inessential LUT can be used to embed a minimum of 1 and a maximum of 2^m bits. For LUTs with a memory size of 16 bits, the maximum stego container value increases from 192 to 2320 bits with an increase in operand size from 4 to 16 bits.

The obtained estimates allow us to state that the volume of stego containers is sufficient to implement the hash sums of 128, 160, 256 and 512 bits, which are commercially used now.

The FPGA systems used in the experiment make it possible to provide steganographic storage of data blocks, the size of which does not exceed 2320 bits.

5. Conclusions and Directions of the Further Research

The paper proposes an approach that allows steganographic embedding of additional data into the FPGA program code. In contrast to the known approaches, it is proposed to do this not by means of equivalent changes, but by means of non-equivalent modifications of the FPGA program code. To implement this approach, it is proposed to identify inessential LUT units in the structure of FPGA systems, which perform the processing of approximate data.

This approach is based on two features of the ICT development: the dominance of hardware support in the processing of approximate data, which is usually performed in floating-point formats, and the orientation of FPGA design to perform complete arithmetic operations. The intersection of these features creates structural redundancy that can be used to organize stego containers in circuits with a LUT-oriented architecture.

A feature of these units is that they are involved only in the calculation of the least significant bits of the result. Since such bits are discarded during rounding, the values of inessential units do not distort the final calculation result. The program code of inessential LUT units is proposed to be used for steganographic embedding of additional data.

Experimental study of the proposed approach made it possible to estimate the volume of the stego container formed by the program code of inessential LUT units. The results of the experiment showed the sufficiency of the volume of the stego container for storing data used in the processes of monitoring the integrity of the FPGA program code.

As further directions of research begun in this paper, the following can be emphasized. This paper investigates the effect of autonomous non-equivalent change in the values of the outputs of LUT units on the result of multiplication. It is of interest to study the effects of joint non-equivalent changes in subsets of LUTs on the result. Thus, the first direction of further research is to estimate the volume of the stego container under conditions of joint non-equivalent changes of subsets of LUT units.

Also of interest is the study of approaches to such a non-equivalent embedding that is not detected by means of on-line testing [45, 46]. Existing on-line testing means are usually based on the residue checking. In the case of checking of the result before the discarding of least significant bits, such checking can detect errors in the operation of the device with a non-equivalent embedding. Because of this, making the non-equivalent embedding invisible to residue checking is the second direction of further research on the topic of this paper.

6. References

[1] M. Bishop, Computer Security, 2nd edn., Addison-Wesley, USA, Boston, 2018.

[2] A.I. Awad, M. Fairhurst, Information Security. Foundations, technologies and applications. The Institution of Engineering and Technology, UK, London, 2018.

[3] J. Katz, Introduction to Modern Cryptography, 2nd Edition, CRC Press, Boca Raton, 2018.

[4] H. Amano, Principles and Structures of FPGAs, Springer, 2018.

[5] F. Kastensmidt, P. Rech (Eds.), FPGAs and Parallel Architectures for Aerospace Applications: Soft Errors and Fault-Tolerant Design, Springer, Cham, Switzerland, 2016.

[6] W. Stallings, Cryptography and Network Security: Principles and Practice, 7th ed., Pearson Education Limited, United Kingdom, Harlow, 2017.

[7] W.A. Conklin et al., Principles of Computer Security, 4th ed., McGraw-Hill Education, 2015.

[8] O. Savenko, A. Nicheporuk, I. Hurman, S. Lysenko, Dynamic signature-based malware detection technique based on API call tracing, CEUR-WS 2393 (2019) 633–643.
[9] Y. Yang, F. Chen, X. Zhang, J. Yu, P. Zhang, Research on the Hash Function Structures and its Application, in Proceedings of International Conference Wireless Personal Communications, 2016.
[10] F. Shih, Digital Watermarking and Steganography: Fundamentals and Techniques. 2nd ed., CRC Press, USA, Boca Raton. 2017.
[11] A. Yahya, Steganography Techniques for Digital Images, Springer, 2018.
[12] M. Nematollahi, C. Vorakulpipat, H. Rosales, Digital Watermarking: Techniques and Trends, Springer, Singapore, 2017.
[13] J. Vacca, Computer and information security handbook, 3rd ed., Morgan Kaufmann, Waltham, Mass, 2017.
[14] K. Zashcholkin, O. Drozd, O. Ivanova, P. Bykovyy, Formation of the interval stego key for the digital watermark used in integrity monitoring of FPGA-based systems. CEUR Workshop Proceedings (CEUR-WS) 2623 (2020), 267–276.
[15] J. Fridrich, Steganography in Digital Media, Cambridge University Press, New York, 2010.
[16] A.I. Awad, M. Fairhurst, Information Security. Foundations, technologies and applications, The Institution of Engineering and Technology, UK, London, 2018.
[17] S. Bosworth, M.E. Kabay, E. Whyne (Eds.), Computer Security Handbook, 6th edition. Wiley, 2014.
[18] K. Zashcholkin, O. Drozd, R. Shaporin, O. Ivanova, M. Drozd, Co-Embedding Additional Security Data and Obfuscating Low-Level FPGA Program Code, in Proceedings of IEEE East-West Design and Test Symposium, EWDTS 2020. doi: 10.1109/EWDTS50664.2020.92251112020
[19] Ching-Nung Yang, Chia-chen Lin, Chin-chen Chang, Steganography and Watermarking, Nova Science Publishers, USA New York, 2013.
[20] K. Zashcholkin, O. Drozd, R. Shaporin, O. Ivanova, Y. Sulima, Increasing the effective volume of digital watermark used in monitoring the program code integrity of FPGA-based systems, in Proceedings of 2019 IEEE East-West Design and Test Symposium, EWDTS 2019.
[21] Synopsys, “DWFC Flexible Floating-Point Overview,” no. August, pp. 1–6, 2016.
[22] J. Ghorpade, J. Parande, M. Kulkarni, A. Bawaskar, GPGPU processing in CUDA architecture, Advanced Computing: An International Journal 3(1) (2012) 105–120.
[23] O. Drozd, K. Zashcholkin, R. Shaporin, J. Drozd, Y. Sulima, Development of ICT Models in Area of Safety Education, in: Proceedings of the 18th IEEE East-West Design & Test Symposium, Varna, Bulgaria, 2020, pp. 115–119. doi:10.1109/EWDTS50664.2020.9224861.
[24] A. Kupin, I. Muzyka, R. Ivchenko, Information technologies of processing Big Industrial Data and decision-making methods, in: Proceedings of the International Conference on Problems of Infocommunications Science and Technology, PIC S and T 2018, 2019, pp. 303–307. doi:10.1109/INFOCOMMST.2018.8632096.
[25] V. Hahanov, E. Litvinova, S. Chumachenko, Green Cyber-Physical Computing as Sustainable Development Model, in: Green IT Engineering: Components, Networks and Systems Implementation, SSDC, vol. 105, pp. 65–86. Springer, Berlin (2017).
[26] O. Drozd, K. Zashcholkin, O. Martynyuk, O. Ivanova, J. Drozd. Development of Checkability in FPGA Components of Safety-Related Systems, CEUR Workshop Proceedings, vol. 2762, pp. 30-42 (2020). URL: http://ceur-ws.org/Vol-2762/paper1.pdf.
[27] G.V. Kondratenko, Y.P. Kondratenko, D.O. Romanov, Fuzzy Models for Capacitive Vehicle Routing Problem in Uncertainty, in: Proceedings of the International DAAAM Symposium "Intelligent Manufacturing and Automation: Focus on Mechatronics & Robotics", Vienna, Austria, 2006, pp. 205–206.
[28] O. Savenko, S. Lysenko, A. Nicheporuk, B. Savenko, Approach for the Unknown Metamorphic Virus Detection, in: Proceedings of the International Conference IDAACS, Bucharest, Romania, 2017, pp. 71–76.
[29] D. Maevsky, V. Kharchenko, M. Kolisnyk, E. Maevskaya, Software reliability models and assessment techniques review: Classification issues, in: Proceedings of International Conference IDAACS, Bucharest, Romania, 2017, pp. 894–899.

[30] T. Hovorushchenko, O. Pavlova, M. Bodnar, Development of an Intelligent Agent for Analysis of Nonfunctional Characteristics in Specifications of Software Requirements, Eastern-European Journal of Enterprise Technologies 1(2) (2019) 6–17. doi: 10.15587/1729-4061.2019.154074.
[31] A. Kupin, A. Senko, Principles of intellectual control and classification optimization in conditions of technological processes of beneficiation complexes, CEUR Workshop Proceedings, 2015, 1356, pp. 153–160. URL: http://ceur-ws.org/Vol-1356/paper_34.pdf.
[32] T. Hovorushchenko, Methodology of Evaluating the Sufficiency of Information for Software Quality Assessment According to ISO 25010, Journal of Information and Organizational Sciences 42(1) (2018) 63–85. doi: 10.31341/jios.42.1.4.
[33] H. Park, Truncated Multiplications and Divisions for the Negative Two's Complement Number System, Ph.D. thesis, The University of Texas at Austin, Austin, USA, 2007.
[34] V. Garofalo, Truncated Binary Multipliers with Minimum Mean Square Error: Analytical Characterization, Circuit Implementation and Applications, Ph.D. thesis, University of Studies of Naples “Federico II”, Naples, Italy, 2008.
[35] O. Drozd, M. Kuznietsov, O. Martynyuk, M. Drozd, A method of the hidden faults elimination in FPGA projects for the critical applications, in: Proceedings of the 9th IEEE International Conference DESSERT, Kyiv, Ukraine, 2018, pp. 231–234. doi:10.1109/DESSERT.2018.8409131.
[36] O. Drozd, I. Perebeinos, O. Martynyuk et. al., Hidden fault analysis of FPGA projects for critical applications, in: Proceedings of the IEEE International Conference TCSET, paper 142, Lviv–Slavsko, Ukraine, 2020. doi:10.1109/TCSET49122.2020.235591.
[37] S. Mittal, A Survey of Techniques for Approximate Computing, ACM Computing Surveys 48(4) (2016) 1–33. doi:10.1145/2893356.
[38] H.B. Kekre, D. Mishra, R. Khanna a.o., Comparison between the basic LSB Replacement Technique & Increased Capacity of Information Hiding in LSB‟s Method for Images, International Journal of Computer Applications 45(1) (2012) 33–38.
[39] Intel Quartus Prime Standard Edition User Guide, 2020. URL: https://www.intel.com/content/dam /alterawww/global/en_US/pdfs/literature/ug/ug-qps-getting-started.pdf.
[40] Intel Quartus Prime Standard Edition Handbook. URL: https://www.intel.com/content/dam/www /programmable/us/en/pdfs/literature/hb/qts/archives/qts-qps-handbook-16.0.pdf.
[41] Cyclone IV Device Handbook. URL: https://www.intel.com/content/dam/www/programmable/us /en/pdfs/literature/hb/cyclone-iv/cyclone4-handbook.pdf.
[42] Ashok P. Nadkarni The TCL Programming Language: A Comprehensive Guide. CreateSpace Publishing, 2017.
[43] Intel Quartus Prime Standard Edition User Guide Scripting. URL: https://www.intel.com /content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qps-scripting.pdf.
[44] Delphi 10 Seattle: Embarcadero. URL: https://www.embarcadero.com/docs/datasheet.pdf.
[45] A. Drozd, M. Lobachev, W. Hassonah, Hardware Check of Arithmetic Devices with Abridged Execution of Operations, in: Proceedings of the European Design and Test Conf, Paris, France, 1996, p. 611. doi:10.1109/EDTC.1996.494375.
[46] A.V. Drozd, M.V. Lobachev, Efficient On-line Testing Method for Floating-Point Adder,” Proc. Design, Automation and Test in Europe, in: Proceedings of DATE Conference, Munich, Germany, 2001, pp. 307–311. doi:10.1109/DATE.2001.915042.